A Rule Set to Detect Interference of Runtime Enforcement Mechanisms*

Somayeh Malakuti, Christoph Bockisch and Mehmet Aksit
Department of Computer Science, University of Twente
P.O. Box 217 7500 AE Enschede, The Netherlands
{s.malakuti, c.m.bockisch, m.aksit}@ewi.utwente.nl

Abstract—**Runtime enforcement aims at verifying the active execution trace of executing software against formally specified properties of the software, and enforcing the properties in case that they are violated in the active execution trace. Enforcement mechanism of individual properties may interfere with each other, causing the overall behavior of the executing software to be erroneous. As the number and the complexity of the properties to be enforced increase, manual detection of the inferences becomes an error-prone and effort-consuming task. Hence, we aim at providing a framework for automatic detection of interferences. As the initial steps to create such a framework, in this paper we first provide formal definitions of an enforcement mechanism and enforcement operators. Second, we define a rule set to detect the interference among properties.**

Runtime Enforcement; Interference Rules; Automatic Interference Detection;

I. INTRODUCTION

Reliability is the ability of a software system to perform its required functions under stated conditions for a specified period [1]. It can be attained via different techniques among which we are interested in applying the runtime enforcement technique [2] to ensure functional correctness of the software. Runtime enforcement is the process of checking whether the active execution trace of software adheres to given properties of the software, and enforcing the properties by modifying the execution trace of the software in case that the properties are not satisfied by the executing software.

Complex software has multiple properties to be verified and enforced at runtime. According to the separation of concerns principle, the properties may be specified individually; hence, they are also verified and enforced individually by most of the existing approaches [2-7]. However, since the enforcement of the properties may interfere with each other the individual verification and enforcement of the properties does not necessarily guarantee the overall correctness of the executing software. For example, assume that to enforce the property *A*, we must invoke the method *m*; whereas enforcement of the property *B* may prevent the execution of *m*. Hence, the enforcement of *B* interferes with the enforcement of *A*, causing the overall behavior of the executing software to be erroneous.

In [8, 9], a language to compose the runtime enforcement mechanisms in a non-interfering way is provided; but, the developer must manually detect the interferences. However, as the number of properties for complex software increase, we believe that manual detection of the interferences become an effort-consuming and error-prone activity for the developer.

In this paper, we aim at providing a framework to automatically detect the interference of enforcement mechanisms. As the initial steps to create such a framework, we provide a formal definition of a runtime enforcement mechanism along with our enforcement operators; and we define a rule set to reason about possible interferences among the properties. The rules can further be used to develop a tool for automatic detection of the interferences.

The rest of this paper is organized as follows. In section II, our running example and the inference problem among the properties are explained. In section III, we provide a formal definition of an enforcement mechanism, our enforcement operators, and our rule set to detect the interference. Finally, section IV discusses the conclusions and future work.

II. PROBLEM STATEMENT

Usually, for complex software multiple properties, which are developed incrementally or even by different groups, must be enforced at runtime. Generally, the enforcement mechanisms of the properties can interfere with each other. In subsection *A* we provide an example of such software; and in subsection *B*, we provide examples of such properties which interfere in complex ways.

A. Running Example

The example software, whose runtime behavior is to be verified, is a media player called MPlayer [10]. MPlayer is composed of several interacting components and is executed as two threads, so-called *UI* and *Core*. Fig.1 provides an overall view of MPlayer components and their interactions in handling the user's request of playing a media file. The component *User Interface* receives the command *Play* along with the media file name from the user; and writes the command in *Command Buffer* by invoking the method *WriteCmd*. The component *MPCore* first reads the buffered command by invoking the

* This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

method *ReadPlayCmd* on *Command Buffer*, then it invokes the method *OpenStream* on the component *Streaming*. Consequently, the component *Streaming* creates a file-handler for the media file and checks that the media file is not corrupted. Afterwards, until the end of the media file is reached, *MPCore* repeats the following five steps: 1) it reads chunks of media file by invoking the method *ReadChunk* on *Streaming*. 2) It separates the video and audio streams of the media chunk. 3) It invokes the method *PlayAudio* on *Audio Processor* to send the audio stream to the audio output. 4) It invokes the method *DecodeVideo* on *Video Processor* to decode the video stream; and 5) it invokes the method *PlayVideo* on *Video Processor* to play the video stream. After playing all media chunks, *MPCore* invokes the method *CloseStream* on *Streaming*.

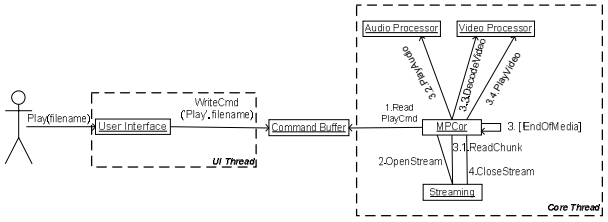

Figure 1. An overall view of MPlayer architecture

B. *Interference of Runtime Enforcement Mechanisms*

Assume that we want to ensure that the requested media file is being played correctly by enforcing the following property at runtime:

- P_1: All of the specified method executions for the thread *Core* must occur according to the specified order in Fig.1; otherwise, the execution of methods which are invoked out of order must be prevented.

To enforce P_1, we make use of an existing runtime verification system, for example MOP [4]. Fig.2 shows a specification of P_1 in the language of MOP. For the sake of brevity some details of the MOP language are omitted. Here, line 1 specifies that the event *ev_ReadPlayCmd* occurs before each execution of the method *ReadPlayCmd*. Likewise, lines 2 to 8 define the events corresponding to the other method executions. Line 9 defines the expected sequence of executions as a predicate in extended regular expression (ERE). Line 10 specifies that the execution of an invoked method must be prevented when the specified extended regular expression is violated.

Assume that the MPlayer functionality is extended to support new video codecs whose video streams often become out of synch with their audio streams. Hence, in addition to P_1, the developer wants to enforce the property P_2.

- P_2: The audio and video streams can become out of synch; however, their timing difference must not be more than ± *d* milliseconds. If this property is violated, playing the media file must be restarted from the beginning.

```
1. event ev_ReadCmd before: execution(* ReadPlayCmd(..)){}
2. event ev_OpenStream before : execution (* OpenStream(..)){}
3. event ev_CloseStream before : execution (* CloseStream(..)){}
4. event ev_NotEoFndOfMedia before : execution (*
NotEndOfMedia (..)){}
5. event ev_ReadChunk before : execution (* ReadChunk (..)){}
6. event ev_PlayAudio before  : execution (* PlayAudio(..)){}
7. event ev_DecodeVideo before  : execution (* DecodeVideo(..)){}
8. event ev_PlayVideo before  : execution (* PlayVideo(..)){}
9. ERE:  ev_ReadCmd  ev_OpenStream (ev_NotEndOfMedia
          ev_ReadChunk ev_PlayAudio ev_DecodeVideo
          ev_PlayVideo)* ev_CloseStream
10. violation { //prevent the execution}
```

Figure 2. A specification of P_1 in MOP

Fig.3 defines P_2 using the raw-specification language of MOP. Line 1 defines and initializes the monitoring variable *d*. Line 2 specifies that the event *ev_DecodeVideo* occurs before each execution of the method *DecodeVideo*. Line 3 specifies that upon occurrence of this event, the method *CalculateDifference,* which is implemented in the component *MPCore*, must be invoked to calculate the difference between audio and video timers. Line 4 verifies whether the difference is within the accepted range of ± *d*; if the difference is not within this range, playing of the current media file is stopped and restarted by invoking the methods *CloseStream* and *OpenStream* in order.

```
1. float d = 40.0;
2. event ev_DecodeVideo before : execution(* DecodeVideo(..)) {
3.    float diff = MPCore.CalculateDifference();
4.    if ( abs(diff) >= d ) {CloseStream(); OpenStream(); }
5.  }
```

Figure 3. A specification of P_2 in MOP

Here, the properties P_1 and P_2 are enforced individually; however, their individual enforcement does not necessarily guarantee the overall correctness of the software. For example, assume that at some point during the execution of MPlayer, the method *DecodeVideo* and all other expected methods before it have been invoked in the expected order; hence according to P_1, the method *PlayVideo* (the event *ev_PlayVideo*) must be invoked (must occur) next. The invocation of *DecodeVideo* also causes the verification of P_2 to start. If the video and audio streams are not synchronized, P_2 invokes *CloseStream* and *OpenStream* and consequently the events *ev_CloseStream* and *ev_OpenStream* occur. However, the occurrences of these events violate P_1 which expects *ev_PlayVideo* as the next event. Hence, the enforcement mechanism of P_1 prevents the executions of *CloseStream* and *OpenStream* and MPlayer keeps playing a media file while the video and the audio streams are not synchronized.

As the complexity and the number of properties increase, the probability that the enforcement mechanisms interfere with each other also increases. Hence, the runtime enforcement tools must provide means to detect the interferences and possibly resolve them. Although several runtime enforcement tools have been developed [2-7], to the best of our knowledge, the

interference of enforcement mechanisms is only elaborated by [8,9] in the security domain. In [8,9], the developer must manually detect the interferences, but a language to define the rules for resolving the interferences is provided. However, as the number of properties for complex software increase, we believe that manual detection of the interferences becomes an effort-consuming and error-prone activity for the developer. Therefore, it is required to investigate on automatic detection of the interferences.

III. Detecting the Interferences of Runtime Enforcement Mechanisms

In this section, first, we provide a formal definition of an enforcement mechanism and our enforcement operators, and then we explain our rule set to detect interferences of enforcement mechanisms.

A. Enforcement Mechanism and Operators

At a high-level of abstraction, we define the execution trace of finite software as a sequence of states $s_0, s_1, s_2, \ldots, s_n$ in which s_i *(i < n)* corresponds to the execution of a method in the software; and s_n depicts end of execution of the software.

A property *P* is a predicate over the states of the software in formalism such as regular expression, temporal logic or propositional logic. Runtime enforcement evaluates the property *P* against the states' changes of the executing software and specifies the next state (i.e. the next method expected to be executed) in the software execution according to results of the verification. An enforcement mechanism *E* for the property *P* is defined as the tuple ($\mathring{a}_f$, $\mathring{a}_o$, *f*, *O*, δ) in which:

- $\mathring{a}_f$ is the finite set of method names over which the predicate *P* is defined; hence, are verified by *E*.
- $\mathring{a}_o$ is the finite set of method names which are enforced by *E*. At least all the members $\mathring{a}_f$ of must be enforced by *E*; hence, $\mathring{a}_o$ is a superset of $\mathring{a}_f$.
- $f \in \mathring{a}_f$ depicts the method which is about to be executed.
- *O* is the finite set of enforcement operators. The elements of this list is chosen from the following possible operators:
  - *CONTINUE*: allows the execution of *f* to be carried out by the executing software. Hence, *E* does not change the execution of the software.
  - *RETURN*: prevents *f* to be executed.
  - *DISPATCH*: prevents *f* to be executed, but invokes $k \in \mathring{a}_o$ instead.
  - *INVOKE*: before executing *f*, it invokes $k \in \mathring{a}_o$; and after the execution of *k*, it continues with the execution of *f*.
  - *HALT*: terminates the execution of the software.
- $\delta: f \rightarrow O \times \mathring{a}_o$ is a function which verifies *f* against the property *P* and enforces an action.

B. A Rule Set for Interference Detection

Assume that *A* and *B* are enforcement mechanisms for the properties P_1 and P_2. We say that *A* interferes with *B*, if *A* violates/validates the property P_2 that is individually satisfied/unsatisfied by the executing software. We reason about the interference of *A* and *B* with the following rules:

- If $\mathring{a}_o^A \cap \mathring{a}_f^B = f$, since the set of enforced methods by *A* is disjoint from the set of verified methods by *B*, there is no interference between *A* and *B*.
- If $\mathring{a}_o^A \cap \mathring{a}_f^B = \varphi$, since the set of enforced methods by *A* is not disjoint from the set of verified methods by *B*, *A* and *B* interfere if the following cases occur for the method $\alpha \in \varphi$:
  - $\exists m \in \mathring{a}_f^A$, $\delta_A (m) \rightarrow (INVOKE, \alpha)$ or $\delta_A (m) \rightarrow (DISPATCH, \alpha)$: Here before the execution of *m*, the enforcement mechanism *A* calls the method α. Assume that the software originally does not contain an invocation to α; hence, *B* verifies α which is invoked from inside *A*. In this case, *A* causes P_2 to be validated.
  - $\exists m \in \mathring{a}_f^A$, $\delta_A (m) \rightarrow (INVOKE, \alpha)$ or $\delta_A (m) \rightarrow (DISPATCH, \alpha)$, and $\delta_B (\alpha) \rightarrow (RETURN, \alpha)$ or $\exists \beta \in \mathring{a}_o^B$, $\delta_B (\alpha) \rightarrow (DISPATCH, \beta)$: Here, *A* calls the method α, but *B* prevents α to be executed; causing P_1 to be violated.
  - $\delta_A (\alpha) \rightarrow (RETURN, \alpha)$ or $\exists \beta \in \mathring{a}_o^A$, $\delta_A (\alpha) \rightarrow (DISPATCH, \beta)$: Here, *A* prevents the method α to be executed; and *B* does not get the chance to verify α. In this case, we cannot precisely determine if P_2 is satisfied or unsatisfied by the software.
- $\exists m \in \mathring{a}_f^A$, $\delta_A (m) \rightarrow (HALT, nil)$: since the execution of software is terminated by *A*, the enforcement of *B* cannot also continue. In this case, we cannot precisely conclude that the executing software satisfies or unsatisfies P_2.

IV. Conclusion and Future Work

In this paper, we discussed the problem of interfering enforcement mechanisms, which causes the overall behavior of the software to be erroneous. As the number and the complexity of properties to be enforced at runtime increases, manual detection of the interferences becomes an error-prone and effort-consuming task for the developers. Hence, we aim at providing a framework for automatic detection of the interferences. As the initial step to have such a framework, we provide a formal definition of a runtime enforcement mechanism and our enforcement operators; along with a rule set to detect the interferences.

As the future work, we would like to develop a tool based on our rule set for automatic detection of the interferences. After detecting the interferences, we must also provide a means to resolve the interferences. Therefore, we aim at proposing a specification language that provides special constructs to

compose the enforcement mechanisms in a non-interfering way.

ACKNOWLEDGEMENT

We acknowledge the feedback from the discussions with our TRADER project partners from Embedded Systems Institute [11].

REFERENCES

[1] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE transaction on dependable and secure computing, vol.1, 2004, pp. 11- 33.

[2] N. Delgado, A. Quiroz Gates, and S. Roach,"A taxonomy and catalog of runtime software-fault monitoring tools," IEEE transactions on software engineering, vol.30. 2004.

[3] M. Kim, M.Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "Java-MaC: a run-time assurance approach for Java programs," Formal methods in system design, vol. 24. Springer-Verlag, 2004.

[4] F. Chen, and G. Rosu, "MOP: an efficient and generic runtime verification framework," OOPSLA, Montreal, Quebec, Canada, 2007.

[5] K. Havelund, "Runtime verification of C programs," TestCom/FATES., LNCS, vol. 5047, 2008.

[6] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to AspectJ," OOPSLA, Oregon, USA, 2005.

[7] J. Ligatti , L. Bauer , and D. Walker, "Edit Automata: Enforcement Mechanisms for Run-time Security Policies", International Journal of Information Security, vol. 4, No 1-2, pp. 2-16. Springer-Verlag, Feb 2005.

[8] L. Bauer, J. Ligatti, and D. Walker, "Composing Security Policies with Polymer,"International Conference on Programming Language Design and Implementation (PLDI*)*, June 2005.

[9] L. Bauer, J. Ligatti, and D. Walker, "Types and Effects for Non-interfering Program Monitors," Theories and Systems, LNCS, vol. 2609, pp. 154-171. Springer-Verlag, November 2003.

[10] MPLayer, http://www.mplayerhq.hu

[11] ESI, http://www.esi.nl